## Suivre van der Pol pour approximer la fonction $\zeta$ de Riemann (Denise Vella-Chemla, mars 2024)

On continue de programmer en utilisant des idées impulsées par la lecture d'un article de van der Pol (cf. lien vers l'article). Dans l'impossibilité d'approximer des intégrales complexes pour l'instant, on intègre et les cosinus et les sinus pour parvenir à nos fins. On voudrait illustrer par ce programme une phrase de Riemann qui interpelle et qui est :

> "*Mais, si l'on changeait cet ordre des termes de la série, on pourrait obtenir pour résultat n'importe quelle valeur réelle.*"

Riemann parle dans cette phrase de l'importance de l'ordre des zéros de la fonction $\zeta$, à prendre par valeurs réelles croissantes, dans les formules qu'il fournit. Sans bien sûr aller jusque là, on voudrait illustrer ci-dessous l'importance de la précision des arguments passés à une fonction, lorsqu'on en cherche des approximations, qui font varier l'exécution d'un programme du tout au tout.

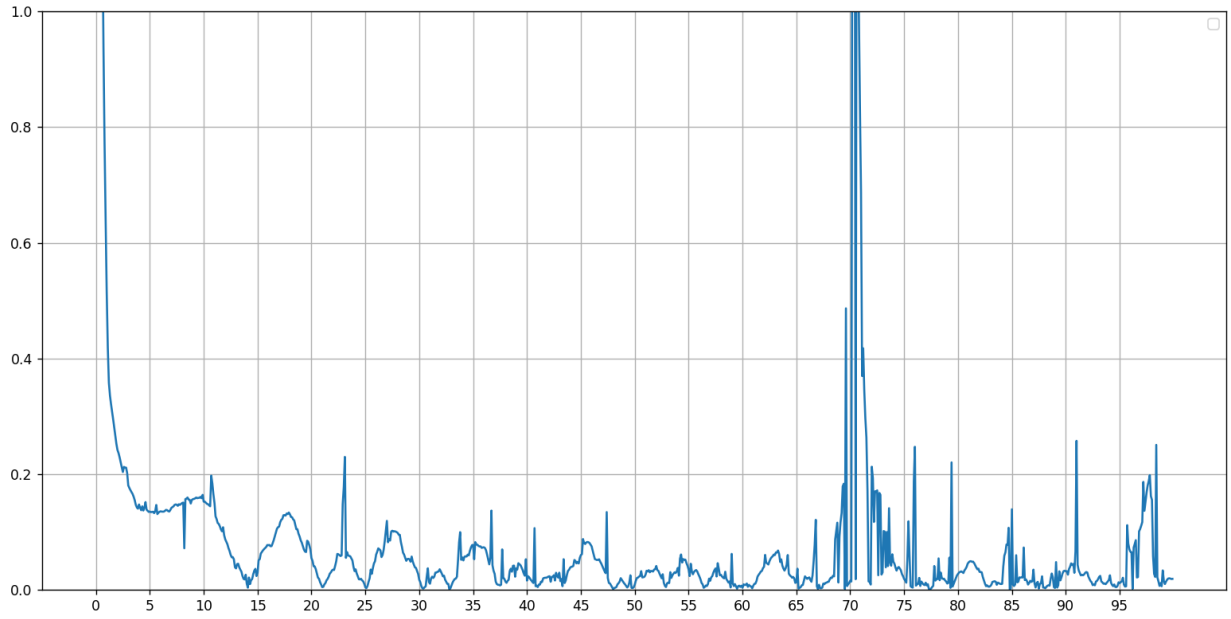
Voyons le programme ci-dessous, on a noté en vert un argument[1] que l'on va faire varier, et on va montrer l'effet de cette variation sur le résultat de l'exécution du programme.

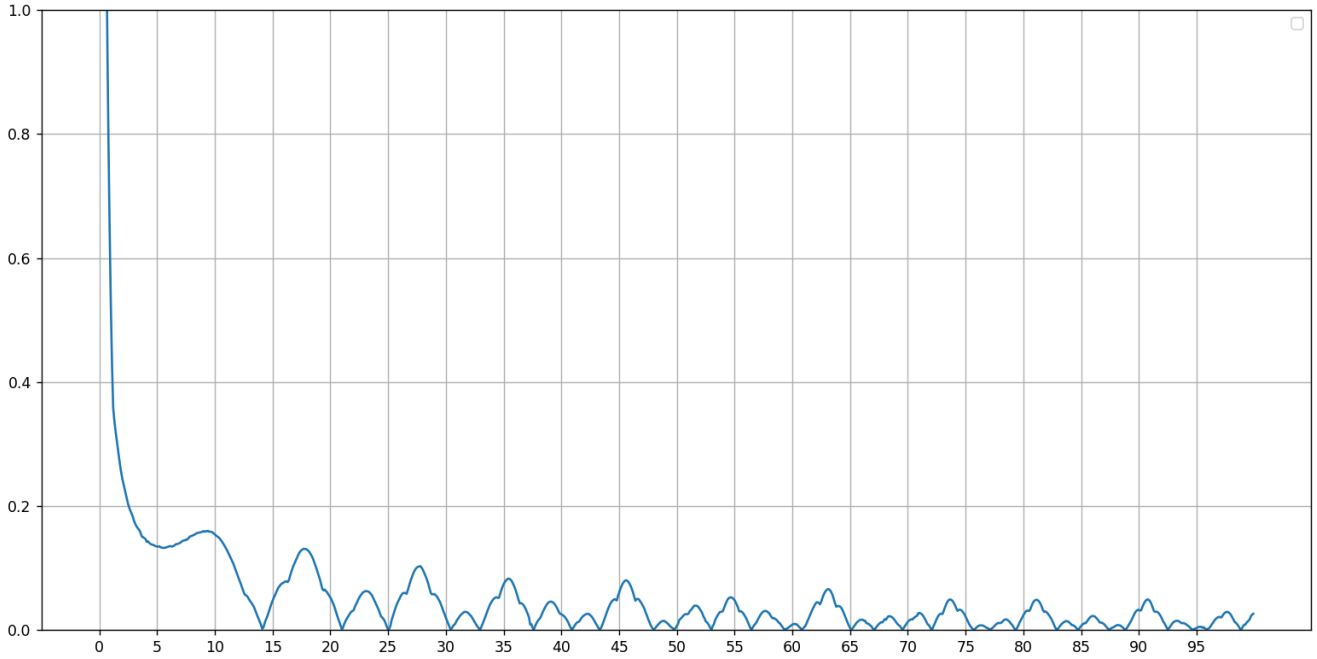
```
from scipy.integrate import quad
import matplotlib.pyplot as plt
import numpy as np
from numpy import sqrt, log, cos, sin, exp, floor, modf, pi

zeros = [ 14.134725142,20,21.022039639,25.010857580,30,30.424876126,32.935061588,
          37.586178159,40,40.918719012,43.327073281,48.005150881,49.773832478,50,
          52.970321478,56.446247697,59.347044003,60,60.831778525,65.112544048,
          67.079810529,69.546401711,70,72.067157674,75.704690699,77.144840069,
          79.337375020,80,82.910380854,84.735492981,87.425274613,88.809111208,90,
          92.491899271,94.651344041,95.870634228,98.831194218,100,101.317851006,
          103.725538040]

def integrale1(fct, a, b, nbpts):    integration par methode des trapezes
     x = np.linspace(a,b,num=nbpts+1,endpoint=True)
     h = (b-a)/nbpts
     y = fct(x)
     return np.sum(y[:nbpts]+y[1:])  * h/2

def integrale2(fct, a, b, nbpts):    integration par methode Simpson 1/3
     assert(nbpts % 2 == 0)
     x = np.linspace(a,b,num=nbpts+1,endpoint=True)
     h = (b-a)/nbpts
     y = fct(x)
     return (y[0]+4*np.sum(y[1:nbpts:2])+2*np.sum(y[2:nbpts-1:2])+y[1:])  * h/3
```

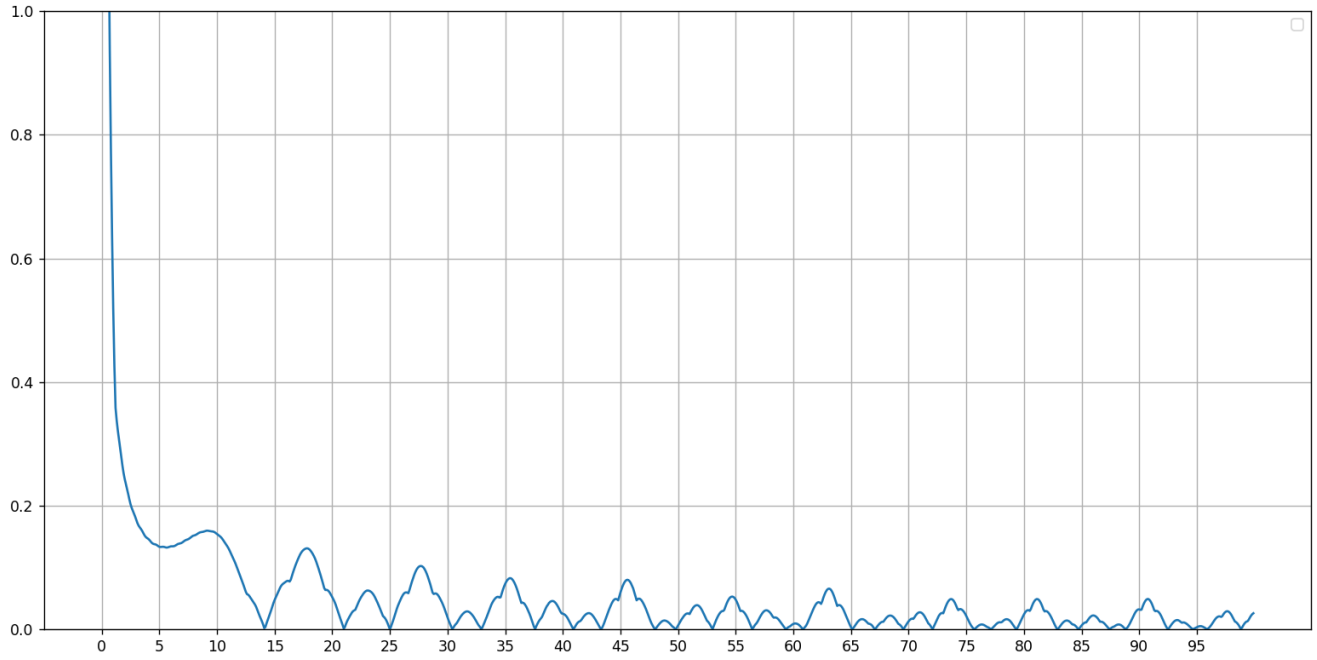
[1] "Argument" est le mot utilisé par les informaticiens pour parler d'une variable passée à une fonction, il ne s'agit pas ici de l'argument d'un nombre complexe.

```
def z(t):
      a = quad(lambda x :  (exp(x) % 1)*cos(x*t)/exp(x/2),-10,10,limit=200)[0]
      b = quad(lambda x :  (exp(x) % 1)*sin(x*t)/exp(x/2),-10,10,limit=200)[0]
      return(sqrt(a*a+b*b))
      return(max(abs(a),abs(b)))

def zp(t):
      a = integrale1(lambda x :  (modf(exp(x))[0])*cos(x*t)/exp(x/2),-10,10,100)
      b = integrale1(lambda x :  (modf(exp(x))[0])*sin(x*t)/exp(x/2),-10,10,100)
      return(sqrt(a*a+b*b))
      return(max(abs(a),abs(b)))

b,c = 100,1000
plt.ylim(0,1)
plt.grid(True)
plt.xticks(range(0,100,5))
plt.legend()
t = [k*b/c for k in range(c)]
plt.plot(t, [abs(zp(tt)) for tt in t])
plt.show()
for z in zeros:
      print('')
      print(z,' --> ',zp(z))
```

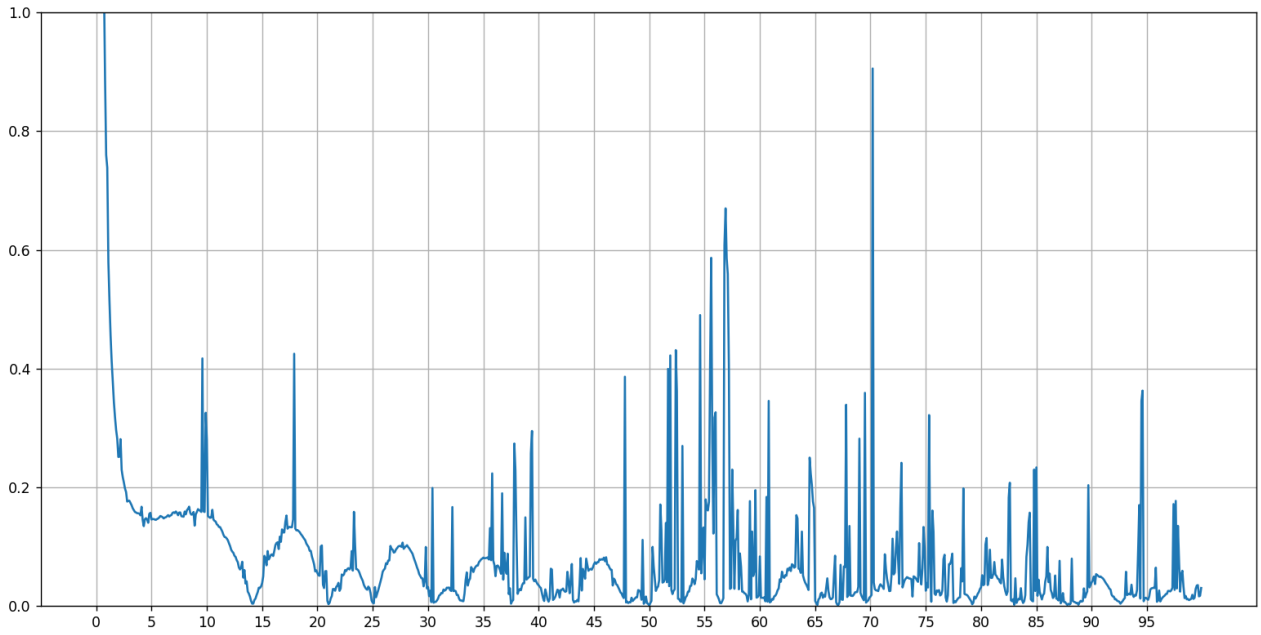
Ci-dessous, les résultats de l'exécution du programme, d'abord avec l'argument 100, puis en remplaçant 100 par 1000, puis par 10000. Le temps d'exécution de la fonction python *quad*, qui calcule une intégrale, quand on souhaite augmenter la précision du calcul, est prohibitif. De plus, on verra en annexe 2 tous les avertissements de l'interpréteur de python qui ne sont pas très encourageants. Enfin, il n'y a pas de différence notable (jusqu'à 100) entre l'argument 1000 et l'argument 10000.



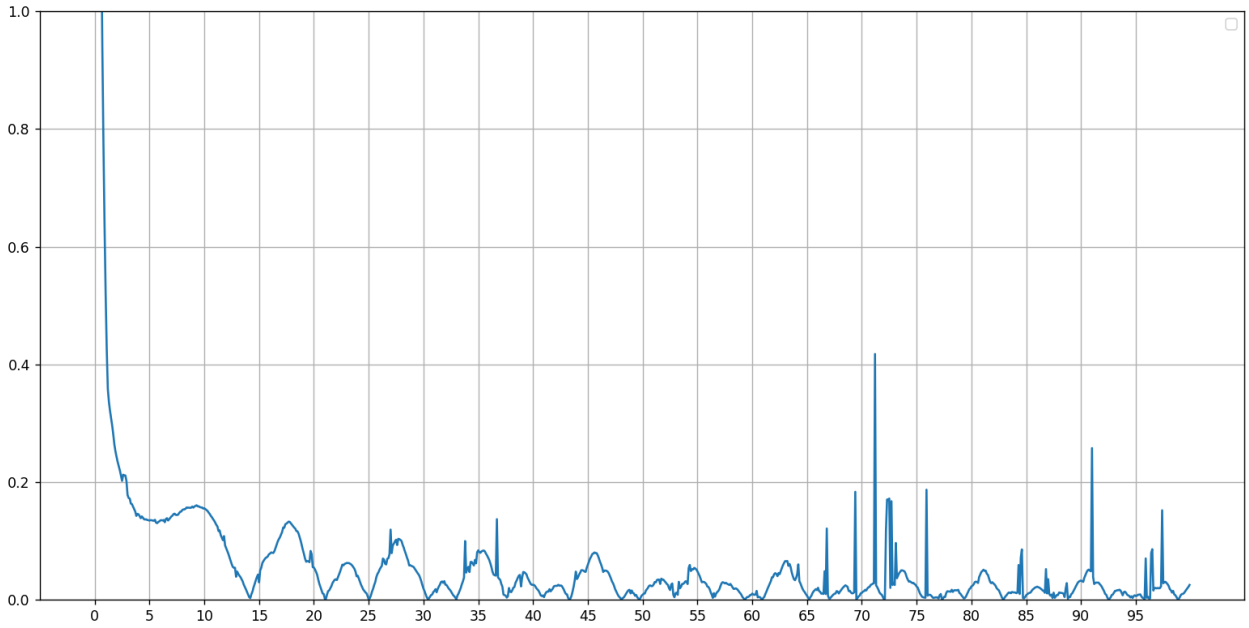



Si l'on remplace les bornes $-10, 10$ par les bornes $-100, 100$, on obtient une fonction pleine de presque-pôles, ainsi :



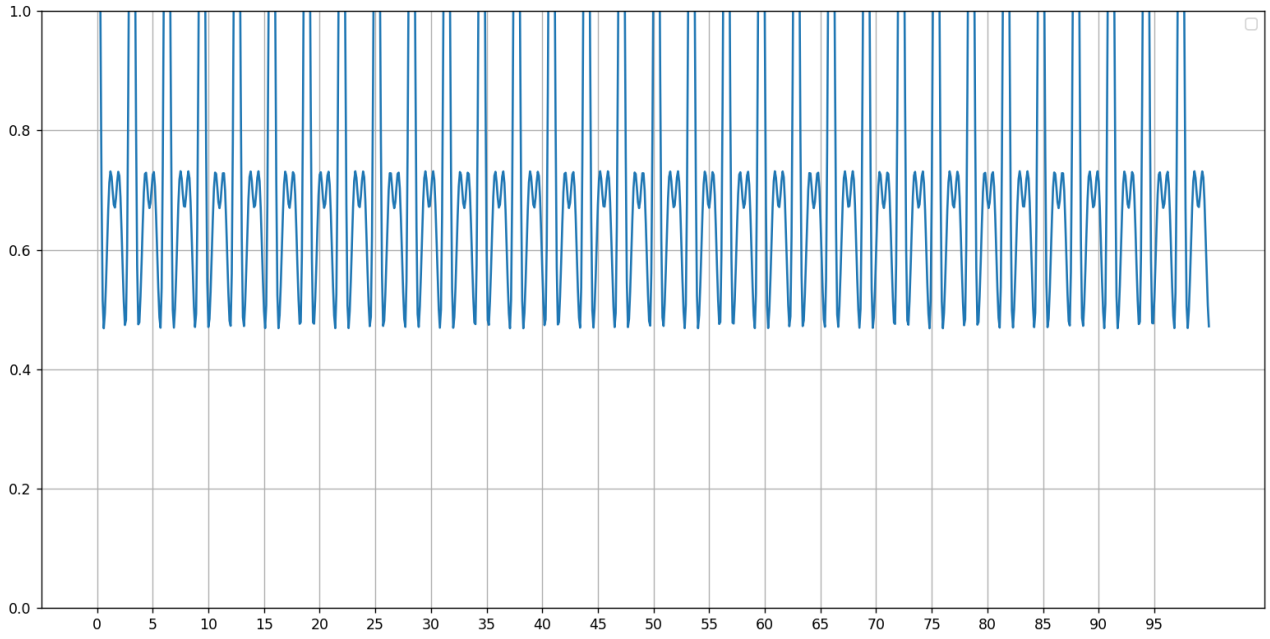

En faisant à nouveau varier un argument, les presque-pôles vont moins haut.

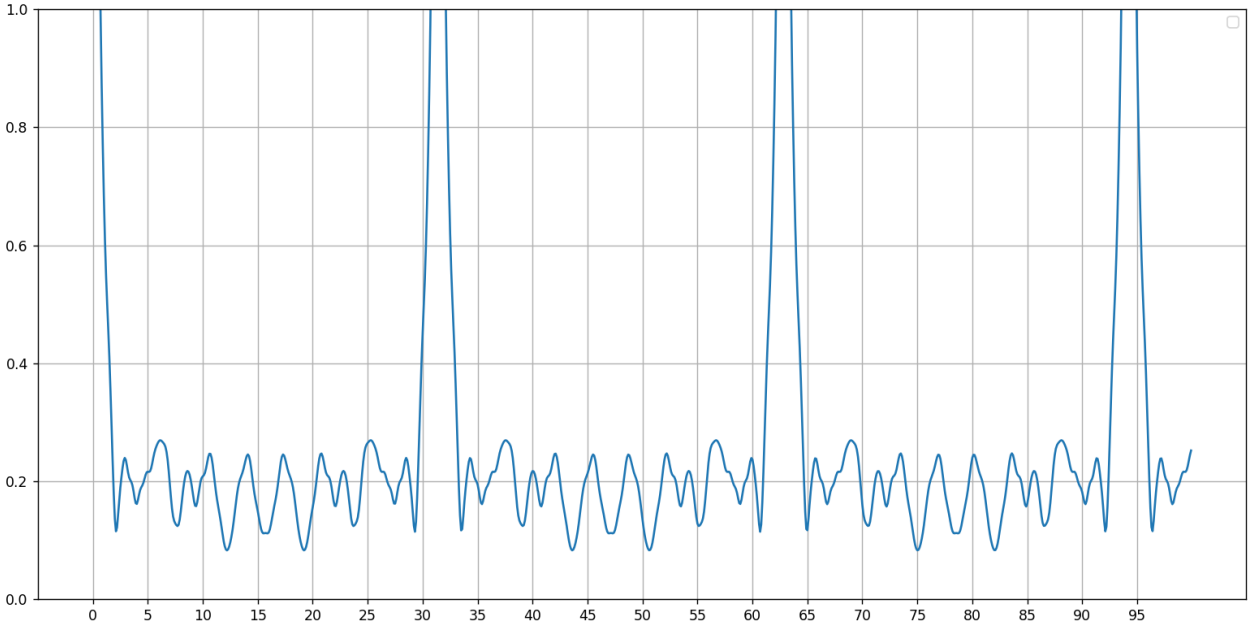


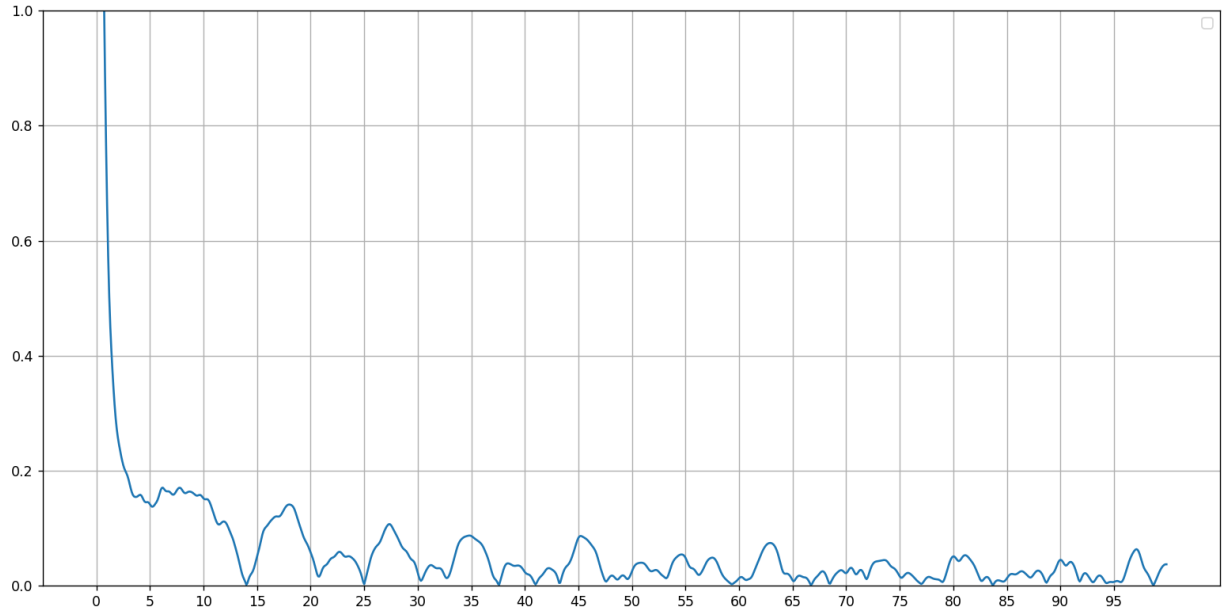

Alors on choisit d'utiliser, par curiosité, ainsi que pour voir si cela permettrait de gagner du temps par rapport à la fonction *quad*, des méthodes d'intégration comme la méthode des trapèzes (fonction *integrale*1 du programme) ou la méthode de Simpson $\frac{1}{3}$ (fonction *integrale*2).

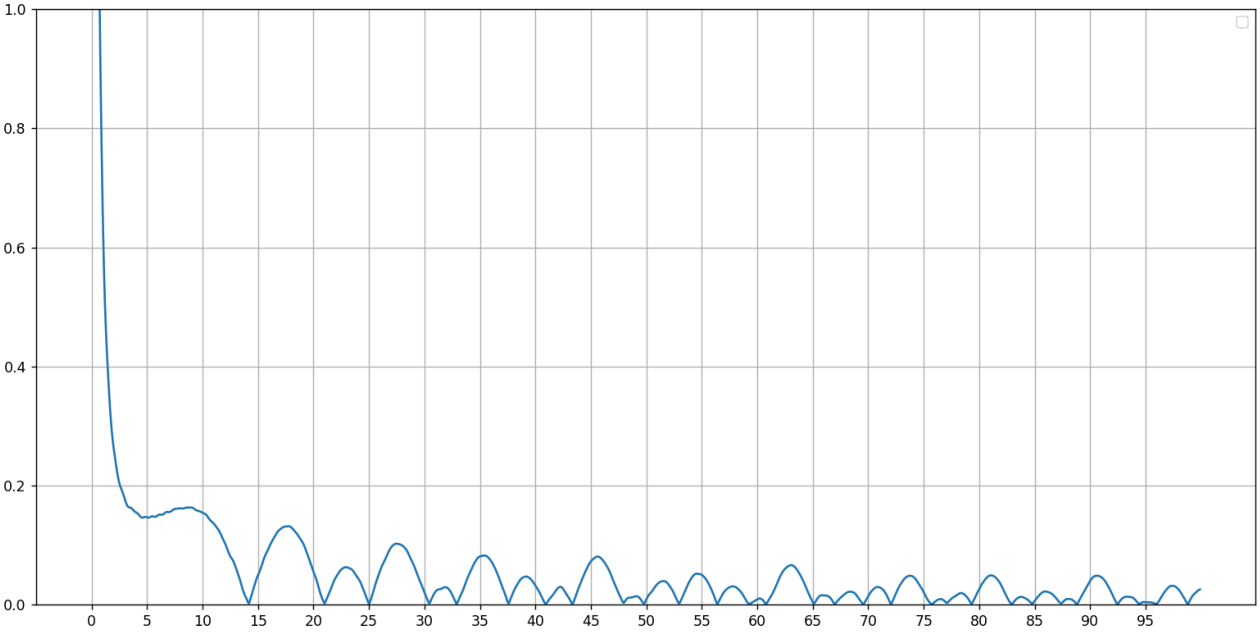
Voici les résultats de l'utilisation de ces fonctions d'approximation des intégrales.

**Méthode des trapèzes pour une précision de 10, 100, 1000 et 10000**

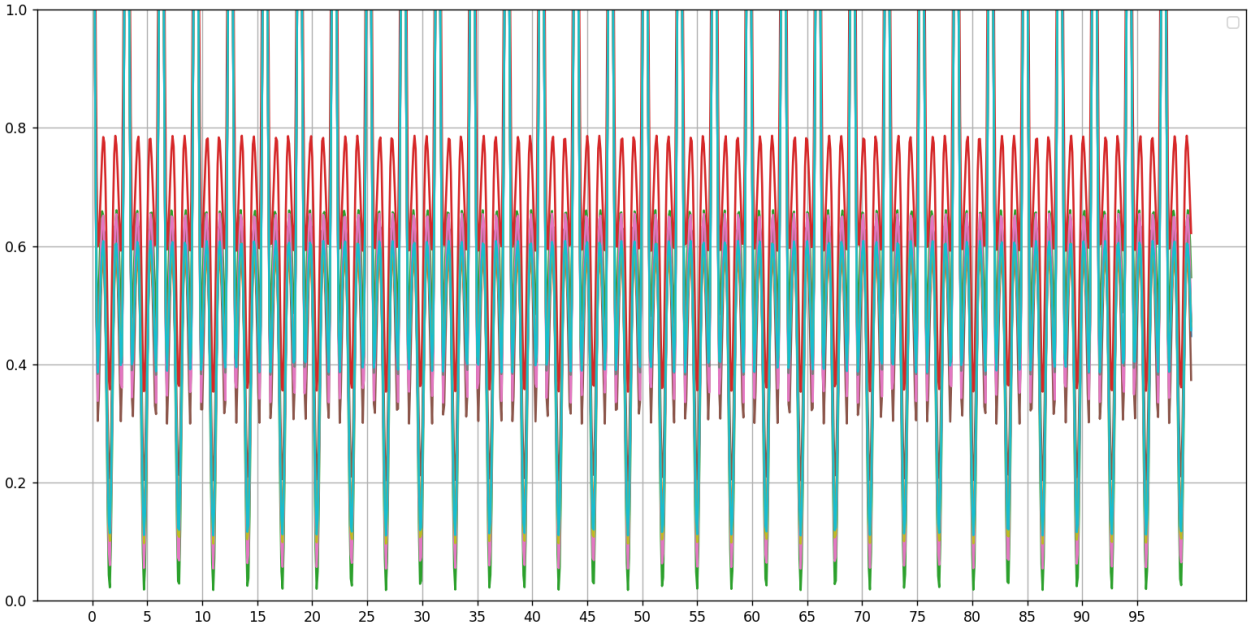


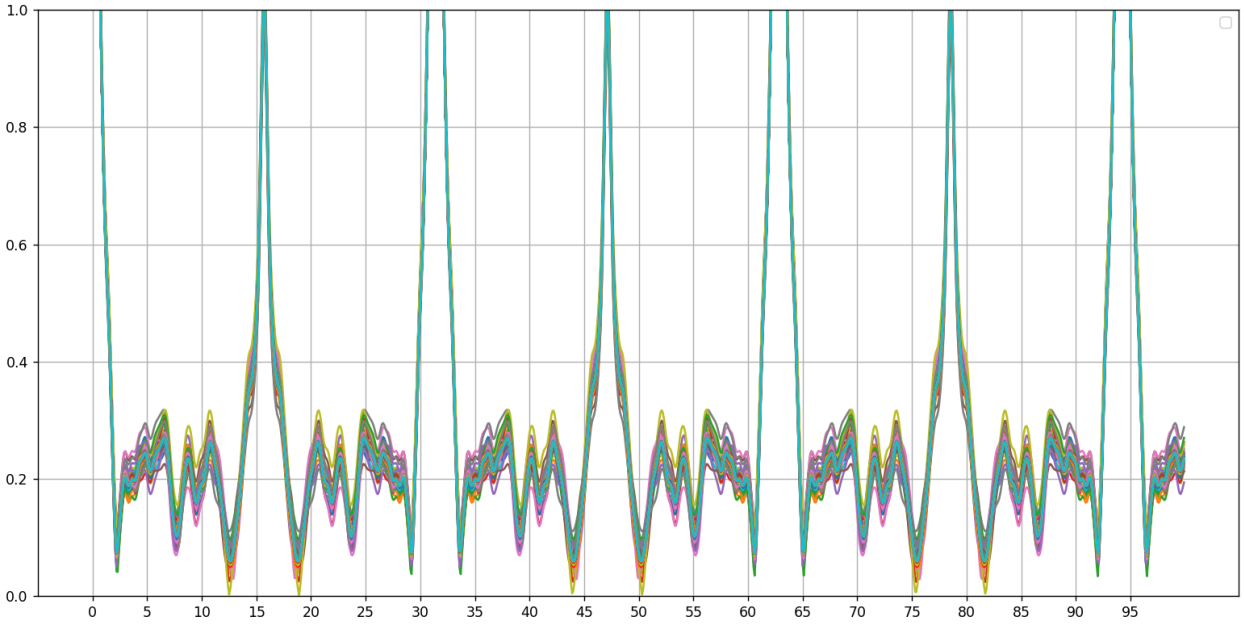



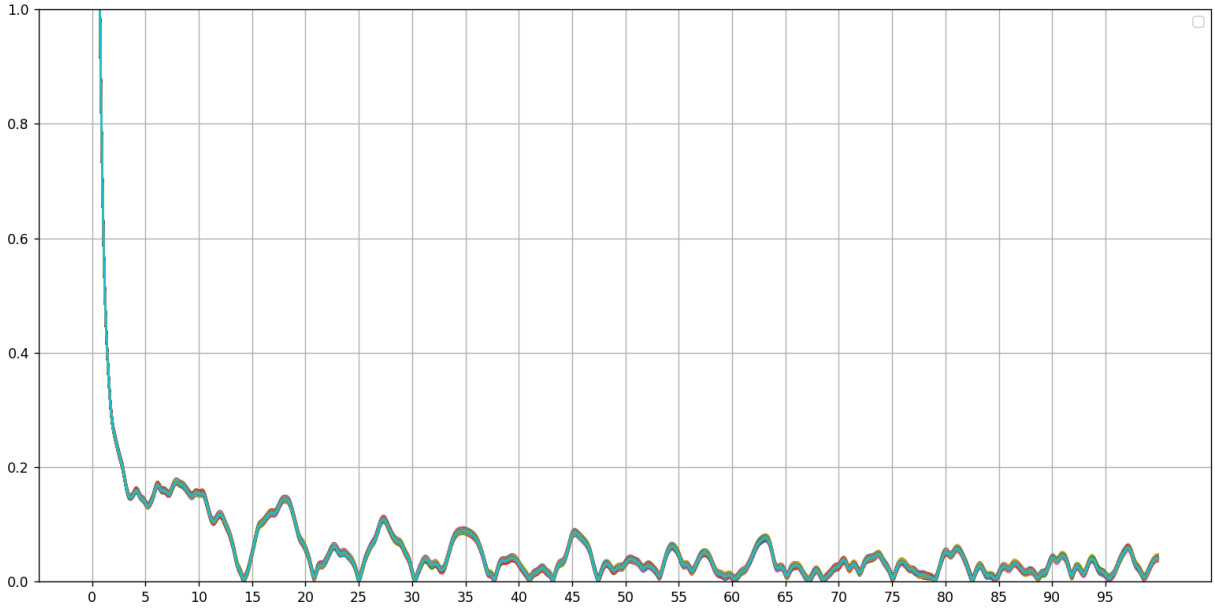

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95

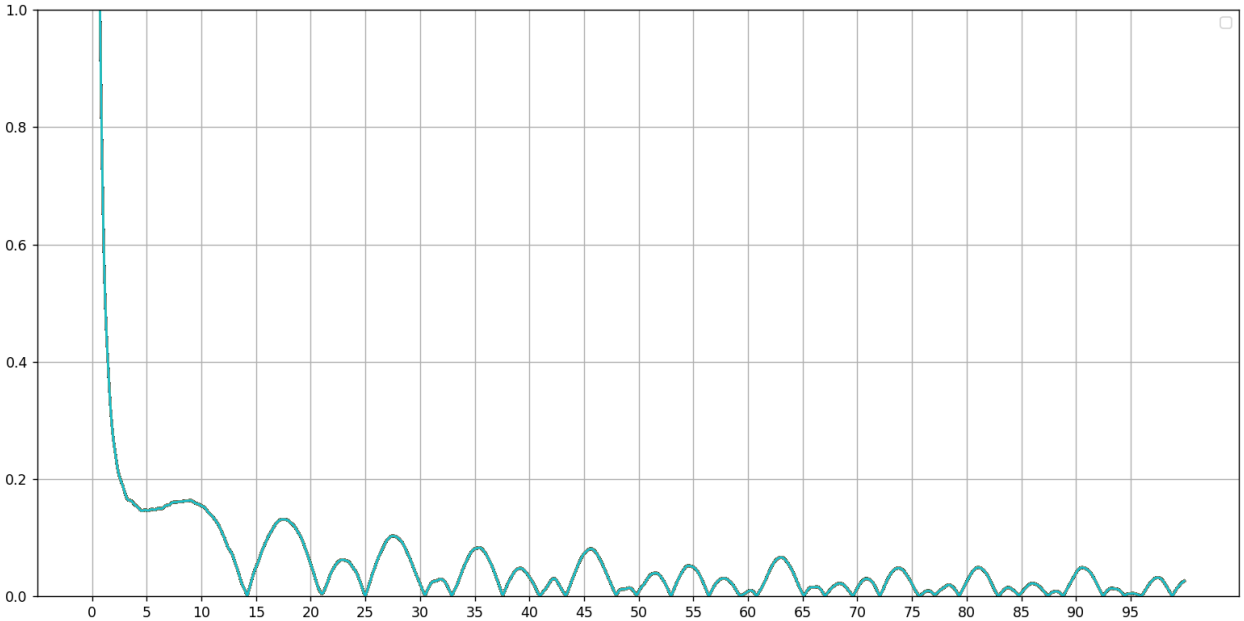

1.0
0.8
0.6
0.4
0.2
0.0
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95

**Méthode Simpson $\frac{1}{3}$ pour une précision de 10, 100, 1000 et 10000**






Enfin, pour appréhender le fait que les zéros de zeta en sont bien localement, on note ci-dessous leur image par la fonction $zp$, ainsi que les images des nombres entiers de dizaines. On voit que l'image d'un zéro est grossièrement plus petite que l'image du nombre entier de dizaines qui lui est inférieur d'un facteur $\frac{1}{10}$ environ (on voit un zéro de plus derrière la virgule dans la partie fractionnaire de l'image). Noter que les nombres 50 et 60 ont des images à deux zéros après la virgule mais tout de même 9 millièmes au lieu de 1 ou 2 pour les zéros qui sont à leur voisinage.

```
14.134725142 --> 0.0016067537523081052
20 --> 0.05532209035993056
21.022039639 --> 0.0026987024591001467
25.01085758 --> 0.0009662281336836151
30 --> 0.020637504764866407
30.424876126 --> 0.0019661521501754863
32.935061588 --> 0.0017667484656647171
37.586178159 --> 0.0018329568813959087
40 --> 0.032272497406023504
40.918719012 --> 0.0010554253153384362
43.327073281 --> 0.0009885789275231932
48.005150881 --> 0.0036477547437878013
49.773832478 --> 0.0018176991302553335
50 --> 0.00907827233423968
52.970321478 --> 0.0019818784620276144
56.446247697 --> 0.0023154470571785015
59.347044003 --> 0.0013514773822215324
60 --> 0.0093078905686516
60.831778525 --> 0.001851124725945632
65.112544048 --> 0.0017173613262055857
67.079810529 --> 0.0029983409048060196
69.546401711 --> 0.0003800330442292773
70 --> 0.014488408386976558
72.067157674 --> 0.0007911817159899152
75.704690699 --> 0.0001459572522272994
77.144840069 --> 0.0035363540815472505
79.33737502 --> 0.0014643212890163874
80 --> 0.02621230979549198
82.910380854 --> 0.0007102674544386193
84.735492981 --> 0.0020297879456133926
87.425274613 --> 0.002697913047415065
88.809111208 --> 0.0004838482658893938
90 --> 0.040102146369664055
92.491899271 --> 0.0009317426688439732
94.651344041 --> 0.003896528887654141
95.870634228 --> 0.0015319154423308876
98.831194218 --> 0.001347237187063686
100 --> 0.02649554592902677
101.317851006 --> 0.001205332943669795
103.72553804 --> 0.000939542425702057
```

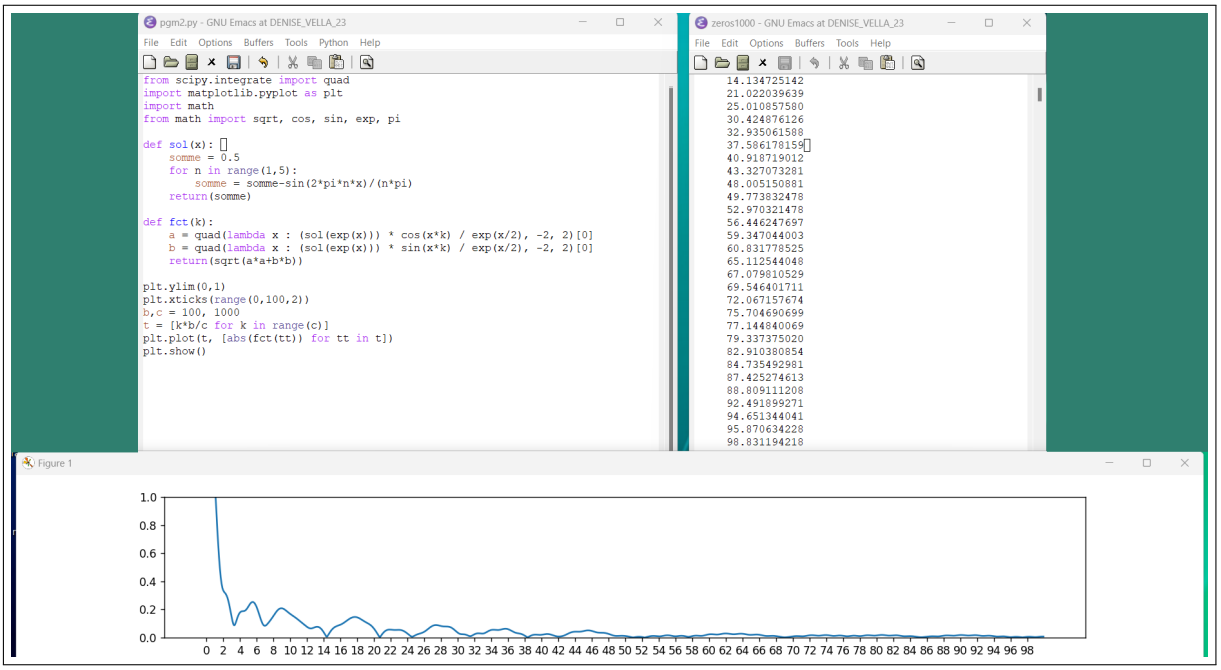
Il semblerait également qu'on ait affaire à un zéro quand il y a une grosse différence entre le cosinus et le sinus des angles considérés. On peut de ce fait utiliser indifféremment (pour $n$ allant jusqu'à 100 et lorsqu'on utilise l'intégrale fournie par le module scipy.integrate de python (notée *quad* dans le programme) la fonction max(|a|, |b|) au lieu de $\sqrt{a^2 + b^2}$. [2]

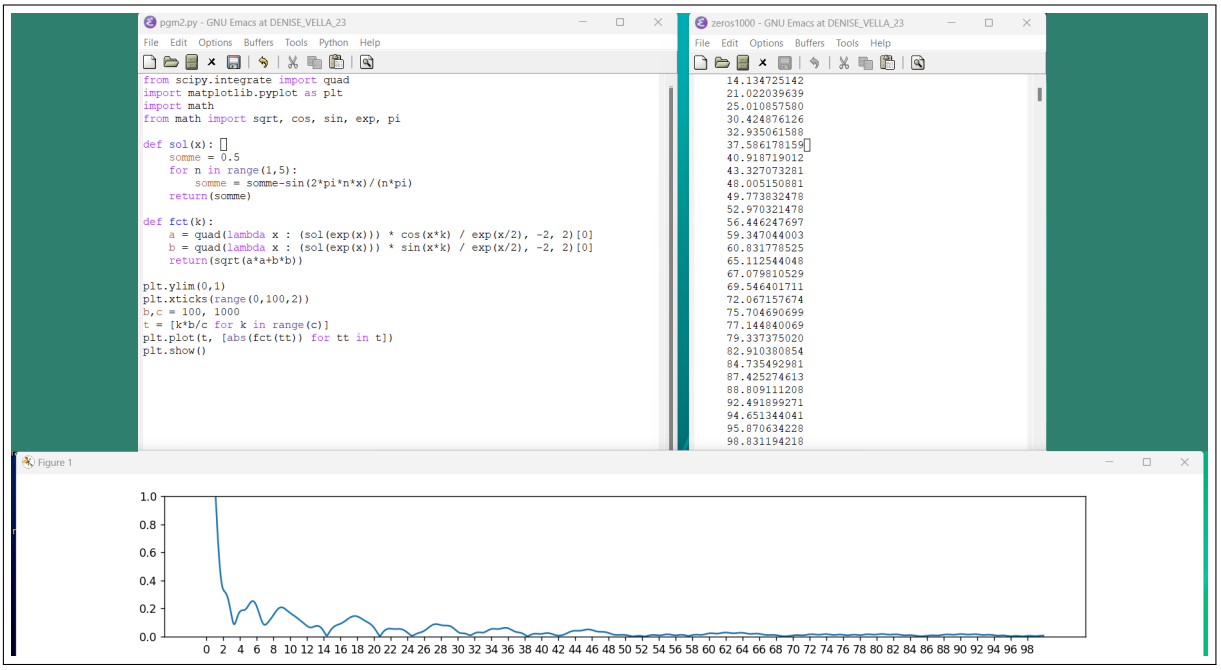
[2] $|x|$ se note abs($x$) en python.

On peut enfin approximer la fonction $f(x) = x \bmod 1$ (qui se note $x$ % 1 en python) définie sur les réels et qui renvoie la partie entière d'un nombre par la fonction $sol(x)$ ci-dessous, si ce n'est qu'au lieu de soustraire une infinité de sinus, on n'en soustrait que 5.

$$sol(x) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{\sin(2\pi n x)}{n\pi}$$

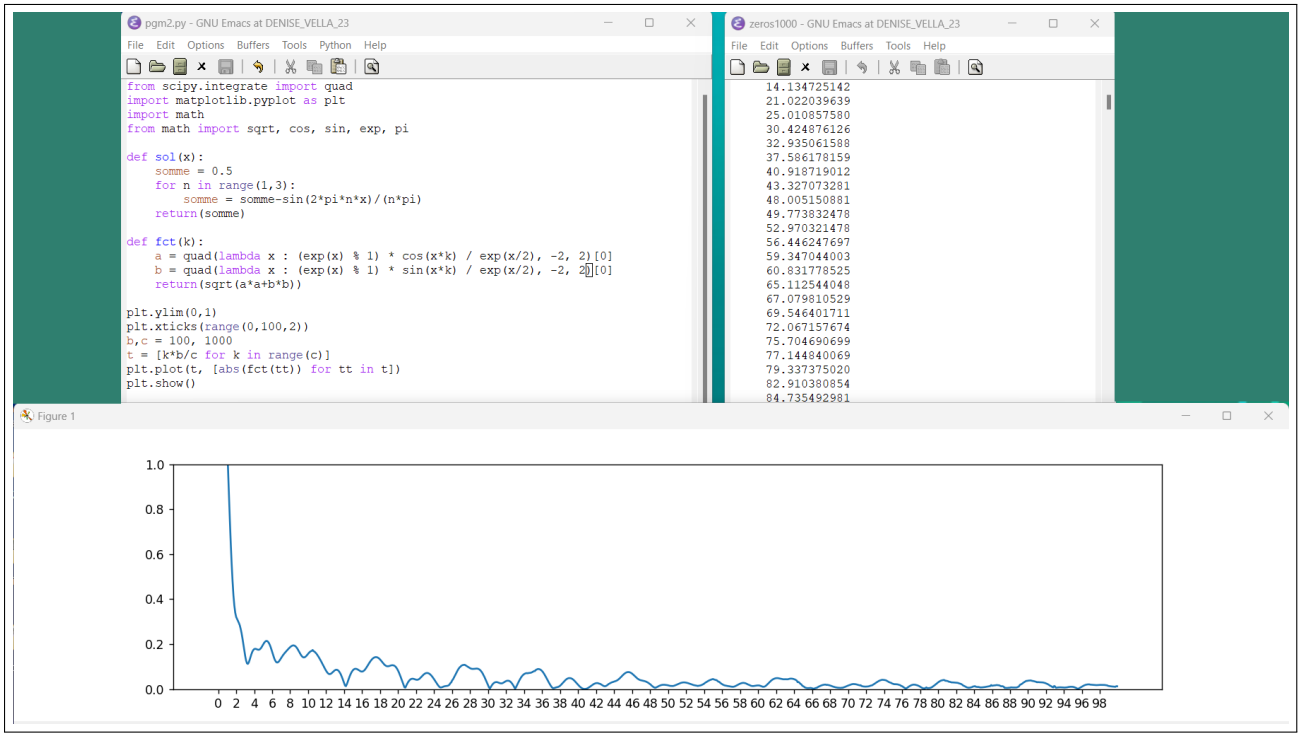
On attrape quelques zéros, mais bien moins, forcément. On essaye en intégrant sur l'intervalle $[-2, 2]$, puis sur l'intervalle $[-3, 3]$.



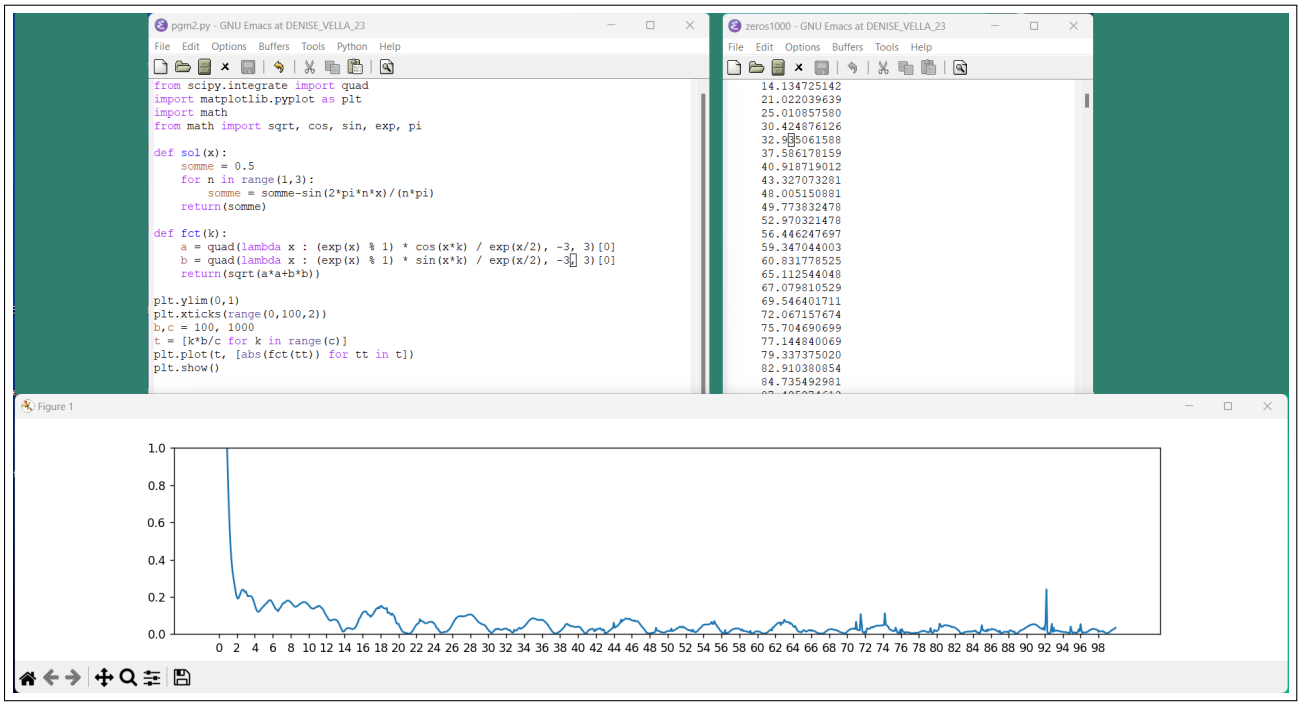



**Annexe 1 : les programmes d'hier soir : les images montrent à la fois, le programme, son résultat, et les zéros de $\zeta$ pour mémoire.**

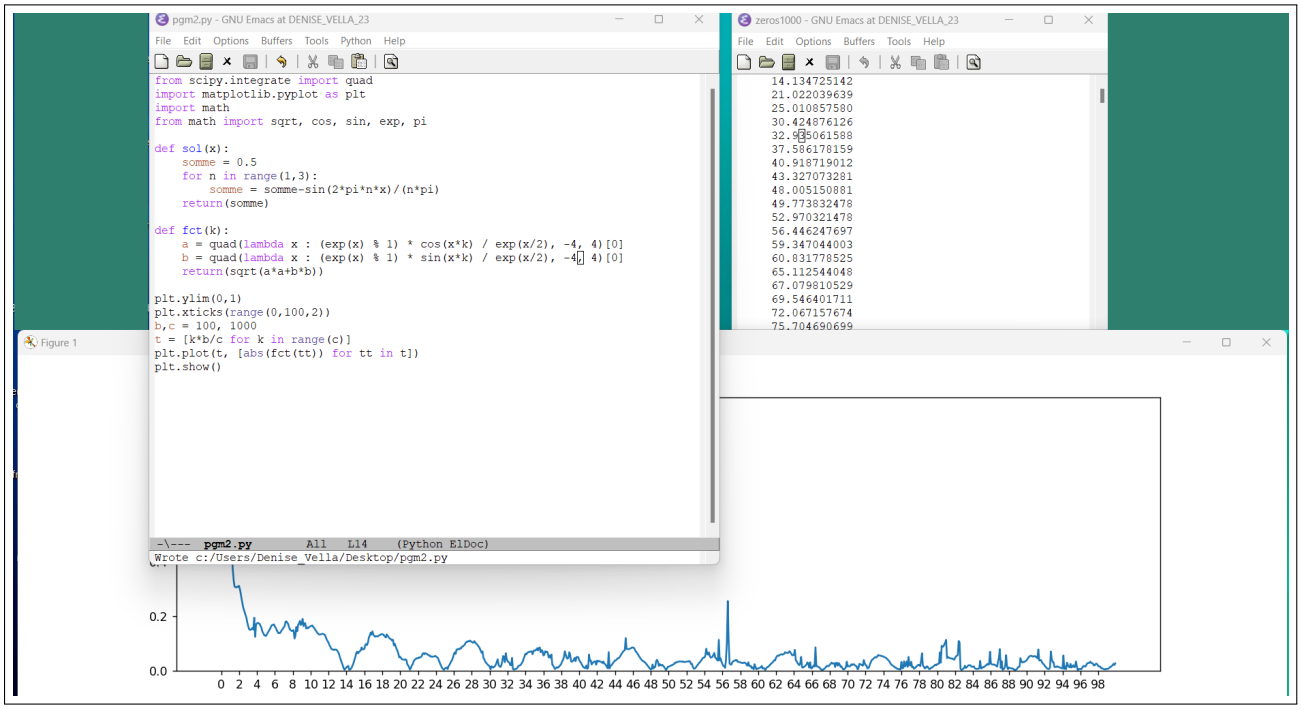
*Attraper 7 zéros en prenant l'intervalle* $[-2, 2]$



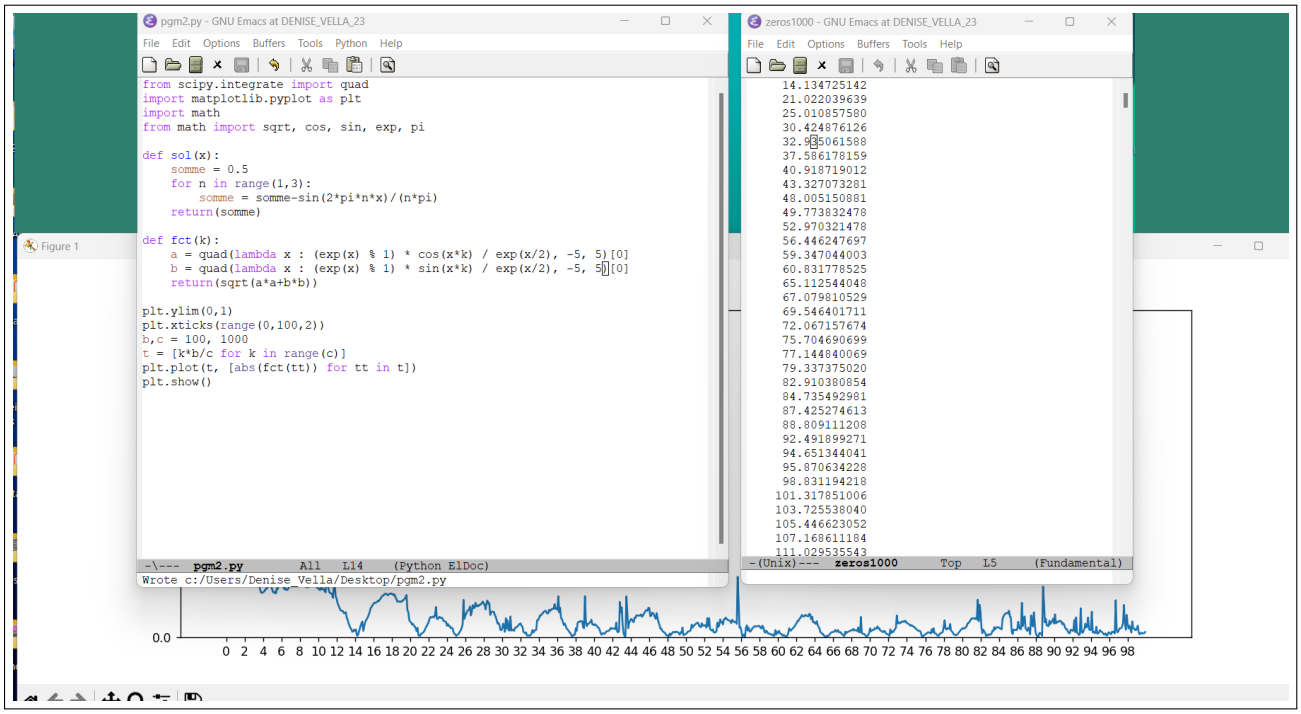

*Attraper 9 zéros en prenant l'intervalle* $[-3, 3]$



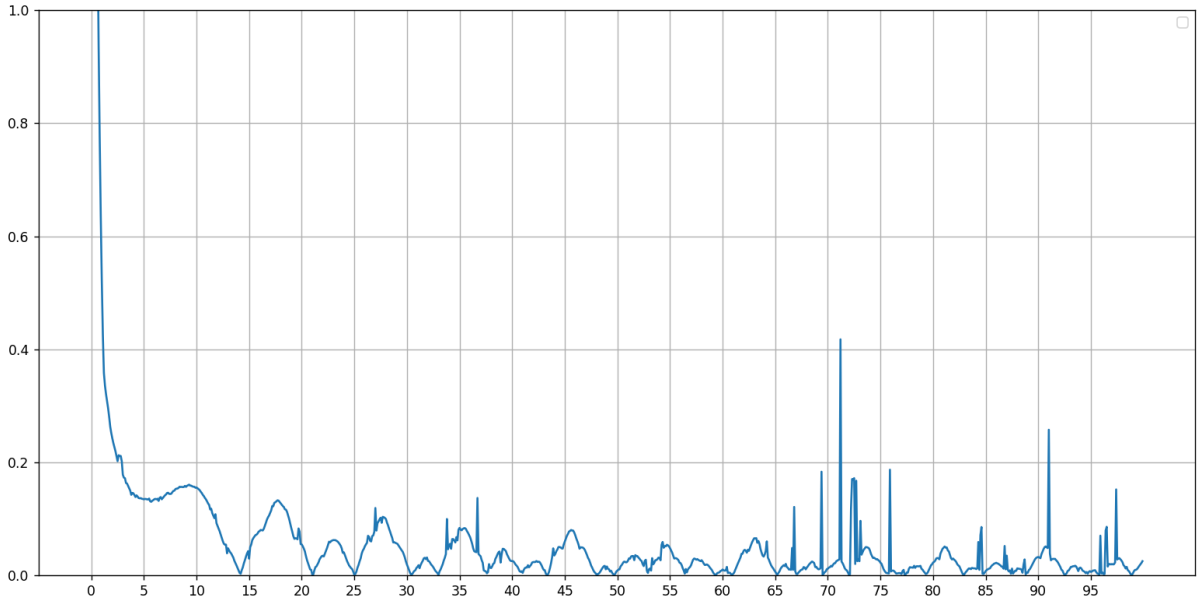
*Attraper 14 zéros en prenant l'intervalle* $[-4, 4]$




*On a perdu le zéro* $52, \ldots$ *en prenant l'intervalle* $[-5, 5]$



*1000 itérations hier mais extra-pôles !*




**Annexe 2 : ce que dit l'interpréteur de python quand il n'est pas content**

```
C:\Users\Denise_Vella\Desktop>python3 integrale.py
No artists with labels found to put in legend.  Note that artists
whose label start with an underscore are ignored when legend()
is called with no argument.
C:\Users\Denise_Vella\Desktop.py:23:  IntegrationWarning:
The maximum number of subdivisions (200) has been achieved.
If increasing the limit yields no improvement it is advised to analyze
the integrand in order to determine the difficulties.  If the position of a
local difficulty can be determined (singularity, discontinuity) one will
probably gain from splitting up the interval and calling the integrator
on the subranges.  Perhaps a special-purpose integrator should be used.
a = quad(lambda x :  (exp(x) % 1)*cos(x*t)/exp(x/2),-10,10,limit=200)[0]
C:\Users\Denise_Vella\Desktop.py:24:  IntegrationWarning:  The
maximum number of subdivisions (200) has been achieved.
If increasing the limit yields no improvement it is advised to analyze
the integrand in order to determine the difficulties.  If the position of a
local difficulty can be determined (singularity, discontinuity) one will
probably gain from splitting up the interval and calling the integrator
on the subranges.  Perhaps a special-purpose integrator should be used.
b = quad(lambda x :  (exp(x) % 1)*sin(x*t)/exp(x/2),-10,10,limit=200)[0]
C:\Users\Denise_VellaDesktop.py:23:  IntegrationWarning:  The
integral
is probably divergent, or slowly convergent.
a = quad(lambda x :  (exp(x) % 1)*cos(x*t)/exp(x/2),-10,10,limit=200)[0]
C:\Users\Denise_Vella\Desktop.py:24:  IntegrationWarning:  The
integral is probably divergent, or slowly convergent.
b = quad(lambda x :  (exp(x) % 1)*sin(x*t)/exp(x/2),-10,10,limit=200)[0]
```